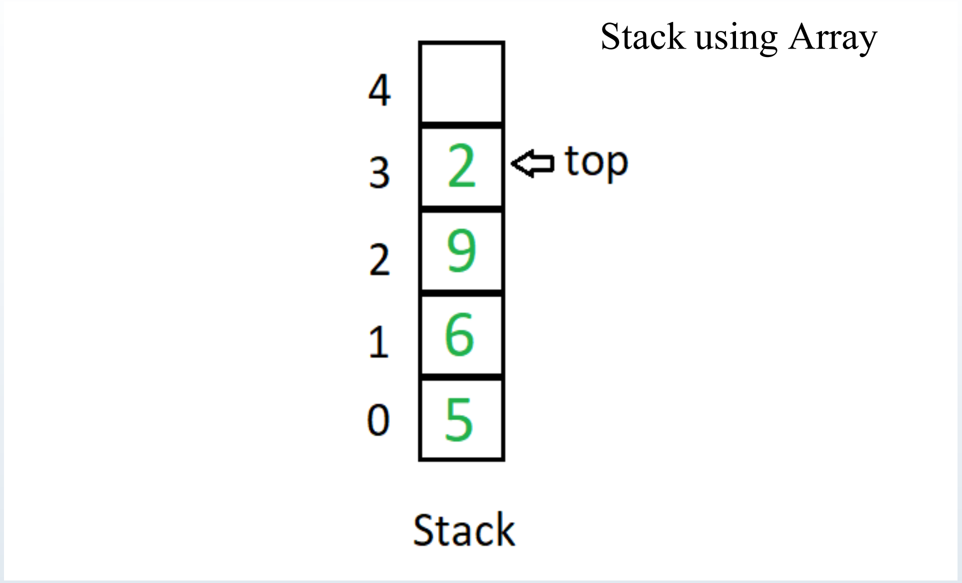
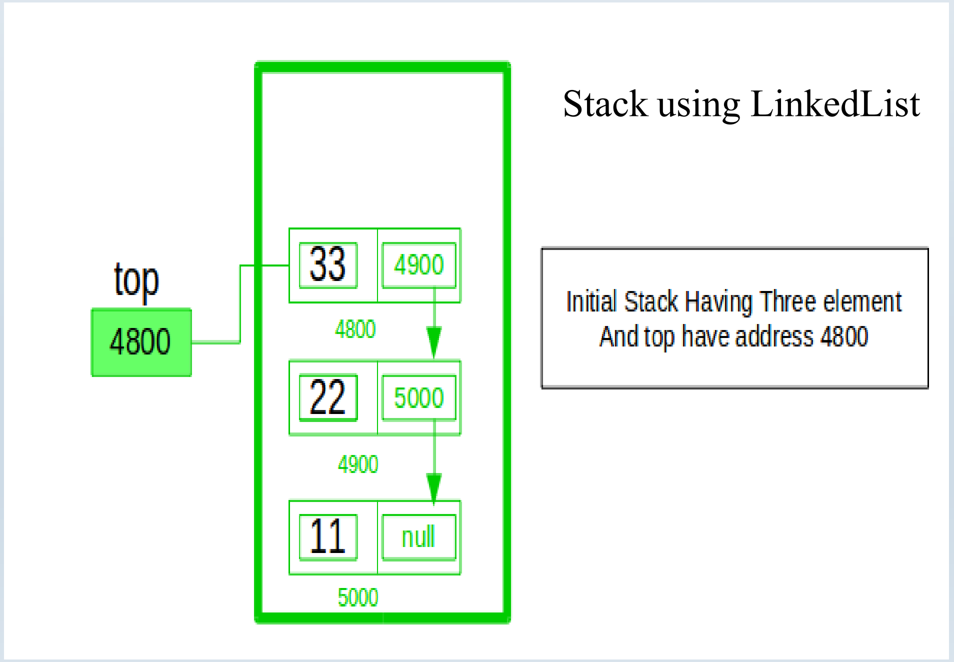
STACK & ITS APPLICATION

Stack

- It’s a linear as well as non-primitive data structure.
- It’s an ordered collection of items that works upon a simple formula called as **LIFO** (Last In First Out).
- Elements are inserted and deleted from one end known as **top**.
- Stack can be a homogeneous/non-homogeneous, static/dynamic data structure.
- Stack can be created using Array and Linked List.
- Stack has 3 operations:
  - Push (insert): check for stack full/overflow
  - Pop (delete): check for stack empty/underflow
  - Display/Peek




Stack using LinkedList
top
4800
33
4900
4800
22
5000
4900
11
null
5000
Initial Stack Having Three element And top have address 4800

Stack using Array

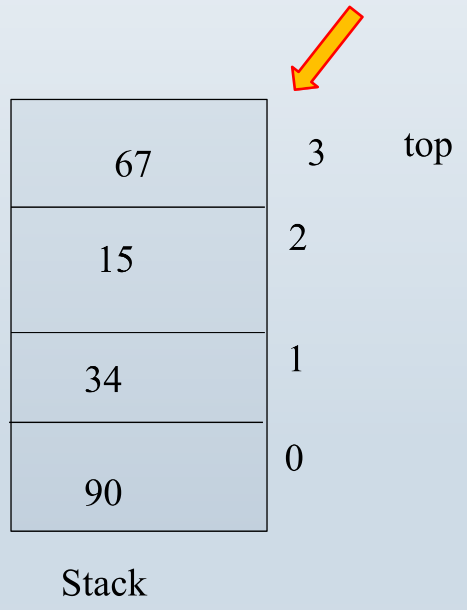
Stack declaration: public static final int MAXSIZE = 4;

int stack [ ] = new int [MAXSIZE]; // stack creation

Stack empty: int top = -1; // top declaration & stack empty

Stack full: if (top = = MAXSIZE - 1)

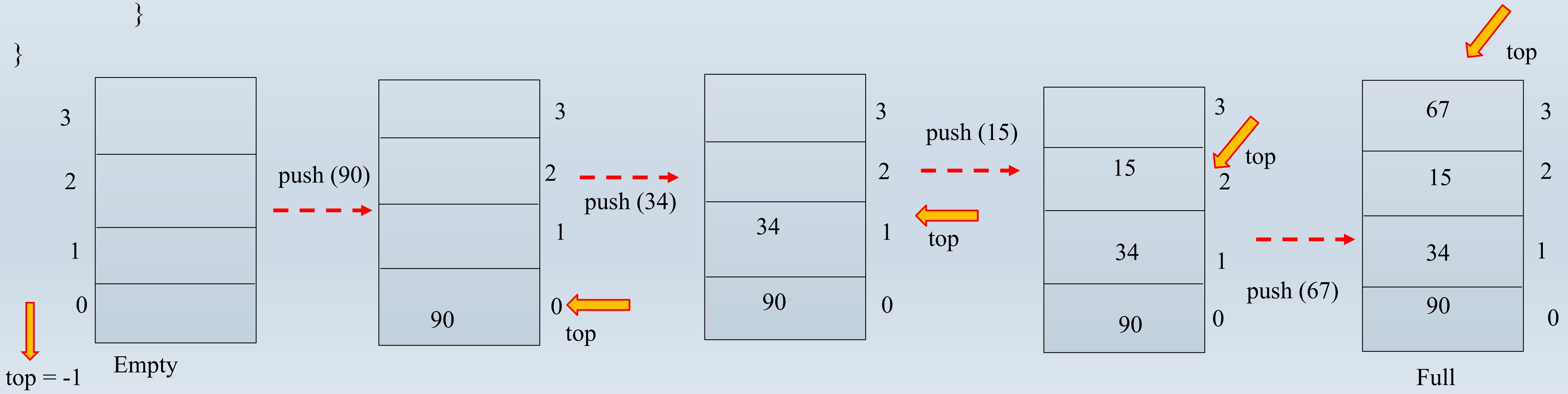
```
void display(int stack[ ], int top)
{
    System.out.println ("Elements present in stack.");
    for (int i = top; i >= 0; i--)
            System.out.println (stack[i]);
}
```



Stack using Array

```
void push (int stack[ ], int top)
{
        if (isFull (top))
                System.out.println ("Stack is Full!");
        else
        {
                Scanner sc = new Scanner (System.in);
                System.out.println ("Enter element");
                int x = sc.nextInt();
                top + +;
                stack [top] = x;
        }
}
```

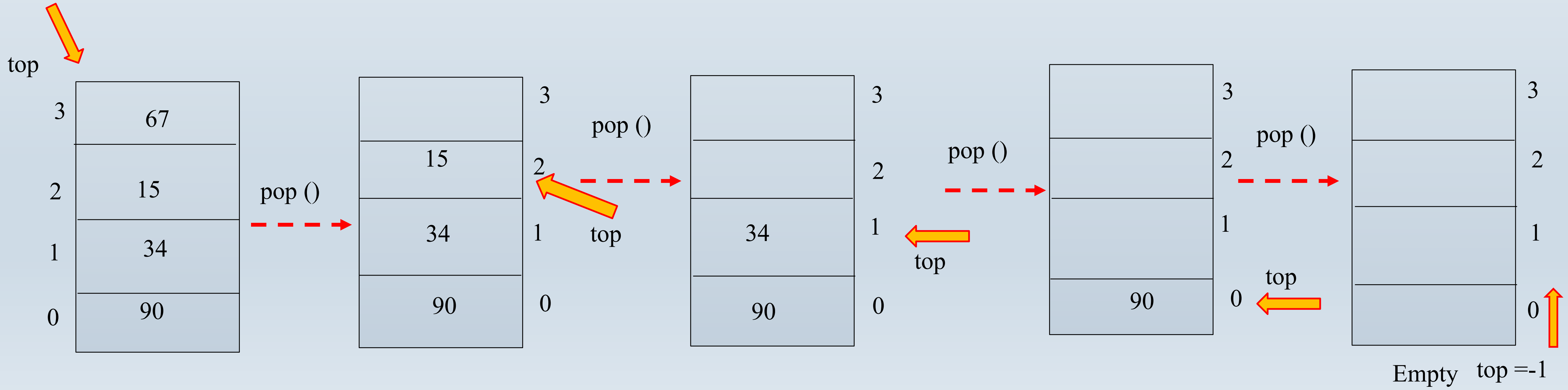
```
boolean isFull (int top)
{
        if (top = = MAXSIZE -1)
            return true;
        else
            return false;
}
```



Stack using Array

```
boolean isEmpty (int top)
{
     if (top = = -1)
               return true;
     else
               return false;
}
```

```
void pop (int stack[ ], int top)
{
    if (isEmpty (top))
        System.out.println ("Underflow!");
    else
    {
        int x = stack[top];
        top - -;
        System.out.println ("Deleted Element is "+ x);
    }
}
```



Stack Questions

Consider a linear last-in-first-out data structure of fixed size say 3, to store the alphabets with the following operations: **pop, push('A'), push('T'), push('K'), push('M'), pop, pop, push('D'), push('W'), push('S')**. Show the value of top after each operation and display the contents of the data structure.

Initially stack is empty, and top = -1

pop----------------------top = -1, stack: (Underflow)

push('A')---------------top = 0, stack: A

push('T')---------------top = 1, stack: A, T

push('K')---------------top = 2, stack: A, T, K

push('M')-------------- top = 2, stack: A, T, K (Overflow)

pop----------------------top = 1, stack: A, T

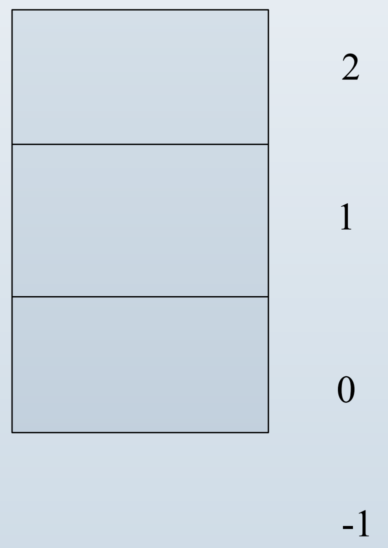
pop----------------------top = 0, stack: A

push('D')---------------top = 1, stack: A, D

push('W')--------------top = 2, stack: A, D, W

push('S')---------------top = 2, stack: A, D, W (Overflow)

Result: top = 2, stack: A, D, W



Stack Questions

Consider a stack of fixed size say 4, to store the alphabets. Initially the stack contains the number 5. The following operations takes place: **PUSH(2),PUSH(9), PUSH(7),POP, POP, PUSH(4), PUSH(3)**. Show the value of top after each operation and display the contents of the stack.

Initially stack contains 5--------------top = 0, stack: 5

push(2)----------------------------------top = 1, stack: 5, 2

push(9)----------------------------------top = 2, stack: 5, 2, 9

push(7)----------------------------------top = 3, stack: 5, 2, 9, 7

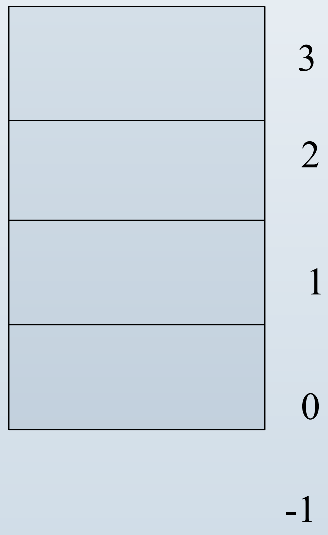
pop-------------------------------------top = 2, stack: 5, 2, 9

pop-------------------------------------top = 1, stack: 5, 2

push(4)---------------------------------top = 2, stack: 5, 2, 4

push(3)---------------------------------top = 3, stack: 5, 2, 4, 3

Result: top = 3, stack: 5, 2, 4, 3



Stack Questions

Consider a stack of fixed size say 4, to store the alphabets. Initially the stack contain the alphabet **’A’**. The following operations takes place: **push(M), push(Z), push(U), push(T), pop, pop, push(S), push(L), push(D)**. Show the value of top after each operation and display the contents of the stack.

Initially stack contains ‘A’-----------top = 0, stack: A

push(’M’)-------------------------------top = 1, stack: A, M

push(’Z’)--------------------------------top = 2, stack: A, M, Z

push(’U’)-------------------------------top = 3, stack: A, M, Z, U

push(’T’)-------------------------------top = 3, stack: A, M, Z, U (Overflow)

pop--------------------------------------top = 2, stack: A, M, Z

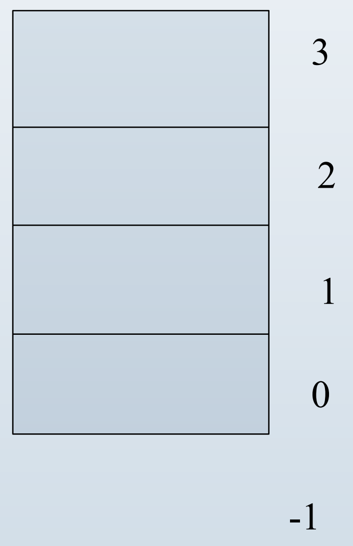
pop--------------------------------------top = 1, stack: A, M

push(’S’)-------------------------------top = 2, stack: A, M, S

push(’L’)-------------------------------top = 3, stack: A, M, S, L

push(’D’)-------------------------------top = 3, stack: A, M, S, L (Overflow)

Result: top = 3, stack: A, M, S, L



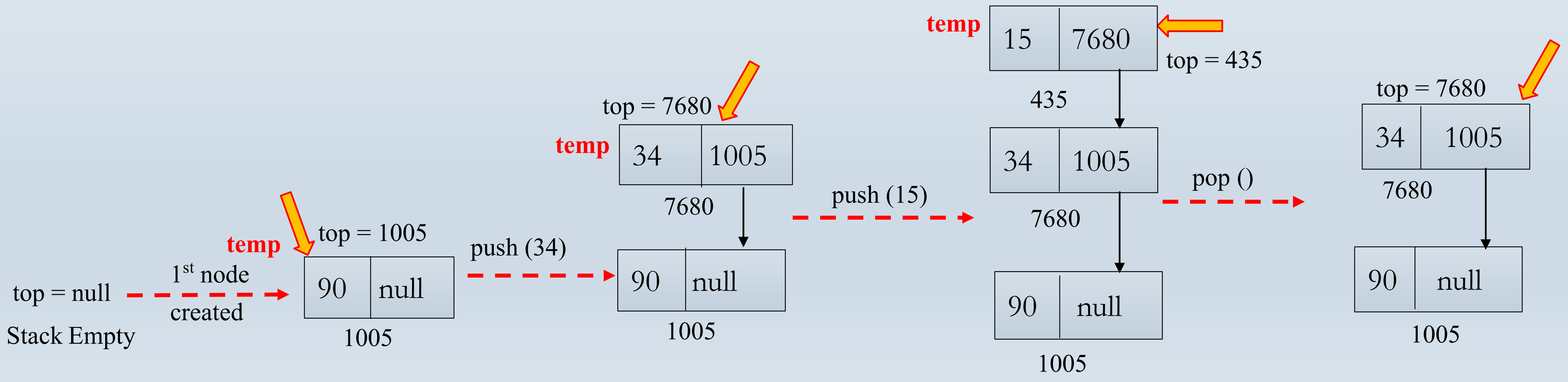
Stack using Linked List

Node declaration:

```
class node
{
      int info;
      node next;
}
```

Stack empty: `node top = null;` // top declaration

No Stack full condition is there except when node creating heap area is full



push(): top increases

pop(): top decreases

Stack using Linked List

```
void pop (node top)
{
    if (top = = null)
        System.out.println (“List Empty");
    else
    {
        System.out.println ("Deleted Element is "+ top);
        top = top.next;
    }
}
void display (node top)
{
        node s = top;
        while (s != null)
        {
          System.out.println (s.info);
          s = s.next;
    }
}
```

```
void push (node top)
{
        Scanner sc = new Scanner(System.in);
        Node temp = new Node();
        temp.info = sc.nextInt();
        if (top = = null)
        {
                top = temp;
                temp.next = null;
        }
        else
        {
                temp.next = top;
                top = temp;
        }
}
```

Application of Stack

- Conversion of Arithmetic Expressions
- Evaluation of Arithmetic Expressions
- Implementation of Recursion/ Function Call

What is Arithmetic Expression?

- An arithmetic expression consists of some operators (+, -, *, /, ^, etc.) and some operands (variables).

  Ex:- ***a + b***

  Here,

  a, b :- operands or variables on which operation is going to be performed

  + :- operator denoting the type of operation going to happen, which is, in this case is Addition (+)

Types of Arithmetic Expressions

- **Infix**:- operator is placed in between the operands.

  Ex:- **a+b**, **a-b**, **a*b**, **a/b**

- **Postfix**:- operator is placed after the operands.

  Ex:- **ab+**, **ab-**, **ab***, **ab/**

- **Prefix**:- operator is placed before the operands.

  Ex:- **+ab**, **-ab**, ***ab**, **/ab**

Example:-

- 2 + (3 - 4) * 8 % 2
- 2 ^ 3 + 7
- 2 ^ 3 ^ 2
- a ^ b / c + d – e
- a+(b+(c-d)*e)/f

Operator Precedence Table

Operator Name & Symbol	Precedence	Associativity
Brackets ( )	4 (Highest)	----
Exponentiation ^	3	Right to Left
Division, Multiplication, Modulus (/, *, %)	2	Left to Right
Addition, Subtraction (+, -)	1 (Lowest)	Left to Right

^ has the highest precedence. Solve 2 ^ 4 (2 to the power 4) = 16. Likewise, 2 ^ 2 = 4.

Both / and * have same precedence, check for associativity. Both have Left to Right associativity. So, go for the part whichever is coming first i.e., 6 * 4 = 24.

Next, solve 12/4 = 3

Both + and - have same precedence and same associativity. So, solve 16 + 24 = 40.

Ex:-

2 ^ 4 + 6* 2 ^ 2 – 12 / 4

= 16 + 6 * 4 – 12 / 4

= 16 + 24 – 12 / 4

= 16 + 24 – 3

= 40 – 3

= 37

Ex:-

7 - 2 * 10 / 5 + 10

= 7 – 20 / 5 + 10

= 7 – 4 + 10

= 3 + 10

= 13

Here, * is coming first before /. So, 2 * 10 = 20.

Next, 20/5 = 4.

Here, - is coming first before +. So, 7- 4 = 3.

Conversion of Arithmetic Expressions

- Conversion of Infix to Postfix
- Conversion of Infix to Prefix

The conversion of these expressions can be done by:

  - Using Stack
  - Without using Stack

Conversion of Infix to Postfix without using Stack

An Infix expression is given and convert it to Postfix expression.

Ex:- A + B * C / D ^ E – F

Since ^ has the highest precedence, convert (D^E) to respective postfix expression (DE^)

= A + B*C/ **DE^** - F

Next, (B*C) will be BC*

= A + **BC*** / **DE^** - F

Next, solve (BC*) and (DE^)

= A + **BC * DE^ /** - F

Next, solve A and (BC*DE^/)

= **ABC * DE^ / +** - F

Finally, solve F and (ABC*DE^/+)

= **ABC * DE^/+F -** **(Final answer)**

*Follow the Operator Precedence Table

Conversion of Infix to Postfix without using Stack

Ex:- A *(B+(C+D)/E)

Since ( ) has the highest precedence, convert (C+D) to respective postfix expression (CD+)

= A * (B + **CD+** / E)

Next, / has the highest precedence, convert (CD+) and E

= A * (B + **CD+E/**)

Next, convert B and (CD+E/)

= A * **BCD+E/+**

Finally, convert A and (BCD+E/+)

= **ABCD+E/+*** (final answer)

*Follow the Operator Precedence Table

Conversion of Infix to Prefix without using Stack

An Infix expression is given and convert it to Prefix expression.

Ex:- A + B * C / D ^ E – F

convert (D^E) to prefix expression (^DE)

= A + B * C / **^DE** – F

Next, solve (B*C)

= A + ***BC** / **^DE** – F

Next, solve (*BC) and (^DE)

= A + **/ * BC^DE** – F

Next, solve A and (/*BC^DE)

= **+ A / * BC^DE** – F

Finally, solve F and (+A/*BC^DE)

= **- + A / * BC^DEF** **(Final answer)**

*Follow the Operator Precedence Table

Conversion of Infix to Prefix without using Stack

Ex:- A *(B+(C+D)/E)

convert (C+D) to prefix expression (+CD)

= A *(B + **+CD** /E)

Next, convert (+CD) and E to (/+CDE)

= A *(B + **/+CDE**)

Next, convert B and (/+CDE) to (+B/+CDE)

= A* **+B/+CDE**

Next, convert A and (+B/+CDE)

= ***A+B/+CDE** (final answer)

*Follow the Operator Precedence Table

Conversion of Infix to Postfix using Stack

STEPS TO FOLLOW

1. Scan the infix expression from Left to Right.
2. If the scanned symbol is an operand, then write it in the Output section.
3. If the scanned symbol is an operator, then consider these below scenario:
   - ✓ If the stack is empty or contains a “(” , then push the scanned operator in the Stack.
   - ✓ If the scanned symbol has higher precedence than the top, then push the scanned operator in the stack.
   - ✓ If the scanned symbol has lower or same priority than the top, then pop the operators from the stack. After that, push the scanned operator in the stack.

5. If the scanned symbol is an ‘(‘, push it to the stack.
6. If the scanned symbol is “)”, then pop the elements from the stack until a “(“ is encountered and discard both the parentheses.
7. Repeat Steps 2-6 until the infix expression is scanned.
8. Print the final output.

Conversion of Infix to Postfix using Stack

Scanned Symbol	Stack	Output
A	Empty	A
*	*	A
(	* (	A
B	* (	AB
+	* (+	AB
C	* (+	ABC
)	*	ABC+
*	*	ABC+*
D	*	ABC+*D
	Empty	ABC+*D*

Ex:- A * (B + C) * D

Output:- ABC+*D*

10 Moves required to solve this problem. Out of which, 4 push operations and 3 pop operations are present.

Conversion of Infix to Postfix using Stack

Scanned Symbol	Stack	Output
A	Empty	A
*	*	A
(	* (	A
B	* (	AB
+	* (+	AB
(	* (+ (	AB
C	* (+ (	ABC
+	* (+ (+	ABC
D	* (+ (+	ABCD
)	* (+	ABCD+
/	* (+ /	ABCD+
E	* (+ /	ABCD+E
)	*	ABCD+E/+
	Empty	ABCD+E/+*

Ex:- A *(B+(C+D)/E)

Output:- ABCD+E/+*

14 moves required to solve this problem. Out of which, 6 push operations and 4 pop operations are present.

Conversion of Infix to Prefix using Stack

STEPS TO FOLLOW

- Reverse the infix expression. Note that while reversing each ‘(‘ will become ‘)’ and each ‘)’ becomes ‘(‘.
- Obtain the postfix expression of the modified expression.
- Reverse the postfix expression to get the desired prefix expression.

Conversion of Infix to Prefix using Stack

Scanned Symbol	Stack	Output
D	Empty	D
*	*	D
(	* (	D
C	* (	DC
+	* (+	DC
B	* (+	DCB
)	*	DCB+
*	*	DCB+*
A	*	DCB+*A
	Empty	DCB+*A*

Ex:- A * (B + C) * D

Reverse:- D * (C + B) * A

Scan this reverse expression from left to right and find the postfix expression.

Finally, reverse the postfix expression to get the desired prefix one.

Output:- * A * + B C D

10 moves required to solve this problem. Out of which, 4 push operations and 3 pop operations are present.

Conversion of Infix to Prefix using Stack

Ex:- (A + B ^ C) * D + E ^ 5

Reverse:- 5 ^ E + D * (C ^ B + A)

Scan this reverse expression from left to right and find the postfix expression.

Finally, reverse the postfix expression to get the desired prefix one.

Output:- + * + A ^ B C D ^ E 5

Scanned Symbol	Stack	Output
5	Empty	5
^	^	5
E	^	5E
+	+	5E^
D	+	5E^D
*	+ *	5E^D
(	+ * (	5E^D
C	+ * (	5E^DC
^	+ * (^	5E^DC
B	+ * (^	5E^DCB
+	+ * (+	5E^DCB^
A	+ * (+	5E^DCB^A
)	+ *	5E^DCB^A+
	Empty	5E^DCB^A+*+

14 moves required to solve this problem. Out of which, 6 push operations and 5 pop operations are present.

Evaluation of Postfix Expression using Stack

STEPS TO FOLLOW

1. Create a stack to store only the operands (or values).
2. Scan the given expression from left to right.
3. If the scanned element is an operand, push it into the stack.
4. If the scanned element is an operator, pop the top two operands from stack. Evaluate the operation and push the result back to the stack.
5. When the expression is ended, the number in the stack is the final answer.

Note:- Let’s assume the top operand is A and next-top operand is B and the operator is /. So, always evaluate the operation like B / A. Similarly, if the operator is *, the operation will be B * A. For -, the operation will be B – A. For +, the operation will be B + A.

Evaluation of Postfix Expression using Stack

Scanned Symbol	Stack	Remark
2	2	2 is pushed
3	2 3	3 is pushed
4	2 3 4	4 is pushed
+	2 7	Pop 4 & 3 to compute 3 + 4 = 7 and push it back
*	14	Pop 7 & 2 to compute 2 * 7 = 14 and push it back
5	14 5	5 is pushed
*	70	Pop 5 & 14 to compute 14 * 5 = 70 and push it back

Ex:- 2 3 4 + * 5 *

Output:- 70

Evaluation of Postfix Expression using Stack

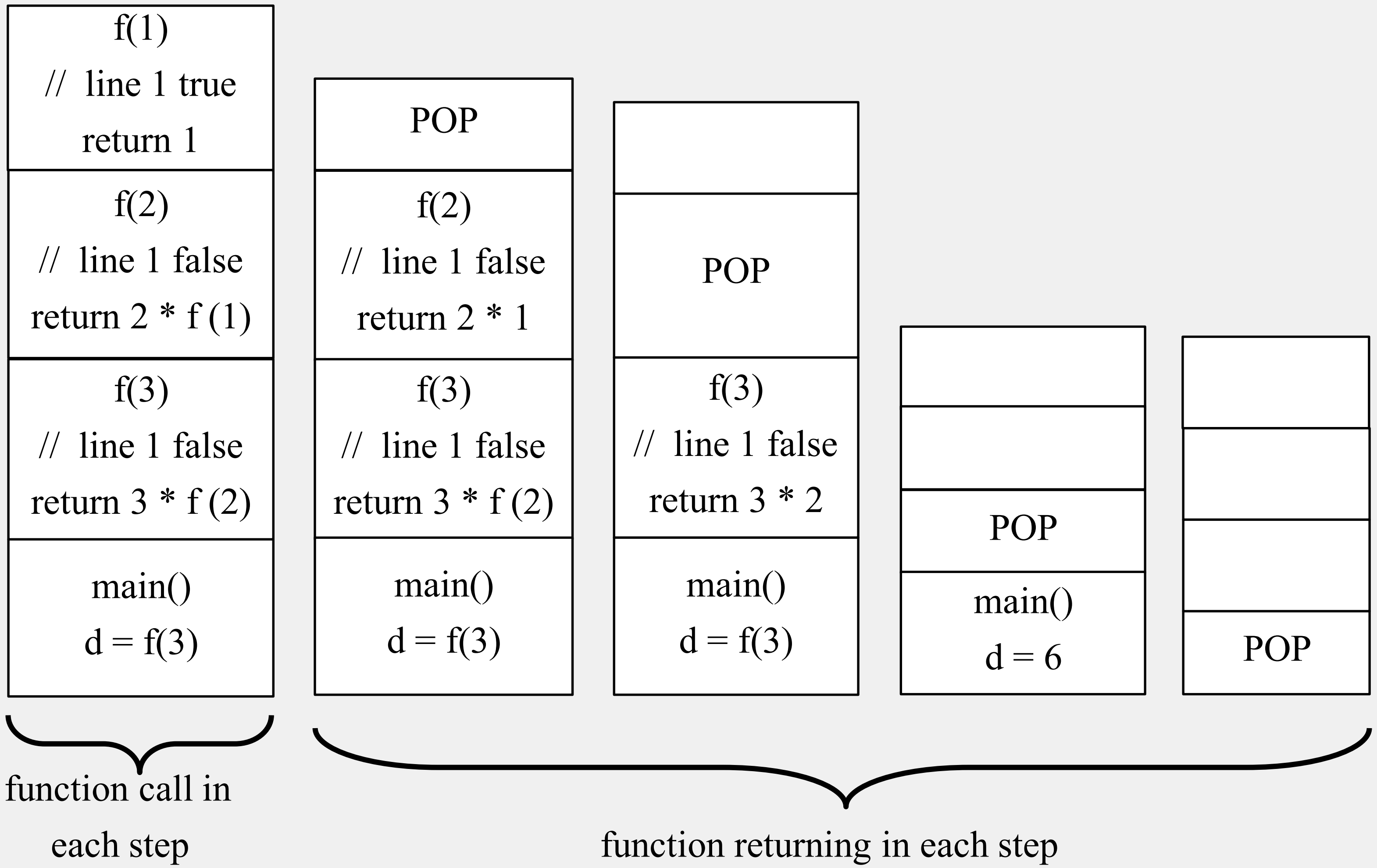
Scanned Symbol	Stack	Remark
2	2	2 is pushed
3	2 3	3 is pushed
1	2 3 1	1 is pushed
*	2 3	Pop 1 & 3 to compute 3 * 1 = 3 and push it back
+	5	Pop 3 & 2 to compute 2 + 3 = 5 and push it back
9	5 9	9 is pushed
-	- 4	Pop 9 & 5 to compute 5 - 9 = - 4 and push it back

Ex:- 2 3 1 * + 9 –

Output:- - 4

Implementation of Recursion using Stack

```
public class Factorial
{
     static int f (int n)
    {
         if (n >= 1)   return 1;    // line 1
         return n * f (n-1);        // line 2
    }
    public static void main (String args[])
   {
       int n = 3;
       int d = f (n);      // main call
       System.out.println (d);
    }
}
```



f(1)
// line 1 true
return 1
f(2)
// line 1 false
return 2 * f (1)
f(3)
// line 1 false
return 3 * f (2)
main()
d = f(3)
POP
f(2)
// line 1 false
return 2 * 1
f(3)
// line 1 false
return 3 * f (2)
main()
d = f(3)
POP
f(3)
// line 1 false
return 3 * 2
main()
d = f(3)
POP
main()
d = 6
POP
function call in each step
function returning in each step